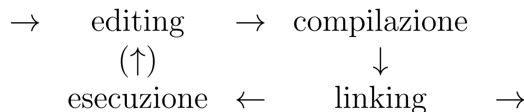
I linguaggi di programmazione possono essere *interpretati* o *compilati*.
Il C è un linguaggio compilato. Le tappe necessarie per produrre un programma funzionante coinvolgono vari passaggi:



Voi scrivete il *sorgente* `primo.c`; il compilatore produce da questo un *file oggetto*, il linker combina tale file con le *librerie* del C e genera un *eseguibile*.
Usando un ambiente integrato, come VisualC++ per Windows, potrete scrivere un programma, generare l' eseguibile e provarlo usando solo comandi da menù. Vedrete traccia delle tappe intermedie in caso di errori: degli errori sintattici si lamenta il compilatore, se il programma è corretto ma chiama una funzione esterna non definita protesta il linker.

Un programma C è composto di *funzioni* che manipolano *dati*. I dati possono essere rappresentati da *costanti* oppure possono essere contenuti in *variabili*.

```
#include <stdio.h>
int main( )
{
  printf("Hi, "); printf("folks!\n");
  return 0;
}
```

Il programma definisce una funzione `main`, che viene invocata con una lista di parametri vuota e restituisce un valore intero (`int`).
Il corpo della funzione `main`, racchiuso fra `{` e `}`, è composto da 3 istruzioni; le prime due chiamano la funzione `printf` passando come argomento due stringhe (racchiuse fra), l'ultima dice che il valore restituito dalla funzione è 0.
`#include <stdio.h>` dice al compilatore di leggere il file *header* `stdio.h` , che contiene informazioni sulla funzione `printf`.

Analizziamo un altro esempio:

```
#include <stdio.h>
#define MIN 1
#define MAX 10

int quadra( int n )
{
    return n*n;
}

int main( )
{
   int i;
   for (i=MIN; i<MAX+1; i = i+1)
       printf("%6d %6d\n", i, quadra(i));
   return 0;
}
```

MIN e MAX sono due *costanti simboliche*;
sono definite le funzioni main e quadra;
main definisce ed usa la variabile intera i ;
printf è chiamata con 3 parametri, e i due campi %6d nel primo argomento indicano come stampare i due argomenti successivi.

Gli **identificatori**, usati come nomi di variabili, di funzioni o di costanti simboliche, sono costituiti da una lettera seguita da lettere o cifre; lettere maiuscole e minuscole sono distinte: raggio e Raggio sono due identificatori diversi. Per convenzione le costanti simboliche si scrivono usando caratteri maiuscoli.
Non possono essere usati come identificatori le seguenti parole chiave:

```
auto       double   int        struct
break      else     long       switch
case       enum     register   typedef
char       extern   return     union
const      float    short      unsigned
continue   for      signed     void
default    goto     sizeof     volatile
do         if       static     while
```

Tipi di dati elementari

char	un singolo byte, 8 bit;
short int	un intero corto (short), 16 bit;
int	un intero, 32 bit;
long int	un intero lungo (long), 64 bit;
float	un reale in singola precisione;
double	un reale in doppia precisione;
long double	un reale in precisione estesa;

Ai tipi char, short, int, long possono essere associati i qualificatori signed e unsigned; una variabile signed short o più semplicemente short può contenere un intero fra -2^{15} e $2^{15}-1$, mentre una variabile unsigned short può rappresentare un intero fra 0 e $2^{16}-1$.
Negli header file <limits.h> e <float.h> sono definite le costanti simboliche che rappresentano i valori minimi e massimi dei vari tipi numerici.

Costanti numeriche

2345 è un int;
123456789L o 123456789l è long;
43890U o 43890u è unsigned int;
i suffissi UL e ul indicano una costante unsigned long.

Espressioni come 23.45 o 2e-3 rappresentano numeri floating-point e sono double;
F o f indicano una costante di tipo float;
L o l come suffisso di un numero decimale indicano una costante di tipo long double.

Un intero che contenga solo le cifre `0 ... 7` e che inizi con la cifra `0` è interpretato come un numero ottale; un intero in notazione esadecimale inizia con `0x` seguito da una sequenza di cifre `0 ... 9, a ... f`; `0112` e `0x4a` sono rispettivamente le scritture ottale e esadecimale di `74`.
I suffissi visti sopra possono seguire un numero intero in qualunque forma.

Costanti carattere
Una *costante carattere* come `'a'` o `'0'` è in realtà un intero piccolo. Ad esempio nel codice ASCII il carattere `'0'` vale 48. Scrivendo `'0'` invece di 48 ci si garantisce una maggiore leggibilità e l'indipendenza dalla codifica.

Alcuni caratteri non stampabili sono rappresentati da una *sequenza di escape*:
`'\n'` è il carattere new-line ;
`'\t'` è il carattere di tabulazione orizzontale;
`'\a'` è il carattere alarm (campanello).

Un carattere può essere espresso anche come `'\ooo'` o `'\xhh'` dove `'ooo'` sono 3 cifre ottali, e `'hh'` sono due cifre esadecimali.
`'\0'` rappresenta il carattere 0.

Stringhe costanti
Una *stringa costante* è una sequenza di 0 o più caratteri racchiusi fra doppi apici (), che non fanno parte della stringa. I caratteri che compongono la stringa possono essere rappresentati da sequenze di escape.

Una stringa non può iniziare su una linea e terminare sulla successiva. Due stringhe consecutive anche su linee diverse sono concatenate in una unica stringa.

Costanti enumerative
Una dichiarazione come

```
enum arrows { UP, DOWN, LEFT, RIGHT };
```

definisce gli identificatori `UP`, `DOWN`, `LEFT`, `RIGHT` attribuendo loro valori interi crescenti a partire da 0. È possibile specificare il valore di una costante in una enumerazione:

```
enum escape {BELL='\a',TAB='\t',NL='\n'};
```

Struttura di un programma

Un programma C inizia con
  direttive (come `#include`, `#define`, ...);
poi seguono
  dichiarazioni di variabili,
  prototipi di funzioni,
e infine
  definizioni di funzioni.

Ogni funzione C ha il seguente formato:

tipo-ritornato nome-funz(lista-param)
{
    dichiarazioni
    istruzioni
}
Le dichiarazioni interne ad una funzione sono a loro volta dichiarazioni di variabili o prototipi di funzioni.

Cominciamo con le dichiarazioni di variabili (ci torneremo dopo aver visto gli altri tipi di dati). Il formato di una dichiarazione di variabile è il seguente:

tipo var1 , *var2 [,...]* ;

dove *var1*, etc., sono identificatori validi. Dove una variabile viene dichiarata, la si può anche inizializzare:

```
char  esc='\033';
float eps=1e-5f;
```

Vari modificatori possono precedere il tipo di una variabile; `const` informa il compilatore che il programma non modificherà le variabili che seguono

```
const char  esc='\033';
const float eps=1e-5f;
```

Operatori ed espressioni

Operatori aritmetici unari: `+` e `-`, binari: `+`, `-`, `*`, `/`.
Si applicano a qualunque tipo numerico.
Se agiscono su operandi dello stesso tipo, il risultato è ancora di quel tipo, per cui `2/3` è una espressione intera costante che vale `0`!
Se gli operandi sono di tipo diverso, l'operando di tipo più ristretto viene convertito al tipo più generale, che è anche il tipo del risultato:
l'espressione `3 * 2.1f` è di tipo `float`.
Per operandi interi è definito anche l'operatore `%` (modulo): `x % y` restituisce il resto di `x` nella divisione per `y`.

Operatori relazionali: `>` , `>=`, `<`, `<=`; hanno tutti la stessa precedenza. Uguaglianza e disuguaglianza sono espresse dagli operatori `==` e `!=`, che hanno precedenza più bassa.
Il valore restituito è 1 se la condizione è vera, 0 se è falsa. Gli operatori relazionali hanno precedenza inferiore a quella degli operatori aritmetici.

Operatori logici:

`!` (negazione): restituisce 0 se l'espressione che segue è diversa da 0, 1 altrimenti.

`&&` (and), `||` (or): `&&` ha una precedenza maggiore di `||`.

Una espressione *expr* data da $expr_1$ `&&` $expr_2$ `&&` ... `&&` $expr_n$ viene valutata nel seguente modo: se $expr_1$ è falsa, le rimanenti sottoespressioni non vengono calcolate, e *expr* vale 0; diversamente, si valuta $expr_2$ `&&` ... `&&` $expr_n$. Analogamente avviene la valutazione di espressioni con `||`.

Operatori bit a bit.

Si applicano solo a operandi interi, con o senza segno.

`~`	NOT	`~0x00E1`	→	`0xFF1E`
`&`	AND	`0x00E1 & 0x0083`	→	`0x0081`
`|`	OR	`0x00E1 | 0x0083`	→	`0x00E3`
`^`	XOR	`0x00E1 ^ 0x0083`	→	`0x0062`
`<<`	left shift	`0x000F << 2`	→	`0x003C`
`>>`	right shift	`0x0040 >> 3`	→	`0x0008`

Operatori di assegnamento.

In C l'assegnamento, indicato da `=`, è un operatore binario.
Questo significa che `a = b` non è una istruzione, come in tutti i linguaggi *normali*, ma una espressione che ha come tipo il tipo di `a` e come valore quello assegnato ad `a`.

```
double a=3.14, c;
int b;
c = 2 * (b = a);
printf("%lf %d %lf\n", a,b, c);
```

stampa i numeri

```
3.140000 3 6.000000
```

Per semplificare la scrittura di espressioni come `i = i+2;` il C fornisce speciali operatori di assegnamento; ad esempio, `i += 2;`, che possiamo leggere come "somma 2 a `i`".

Sono disponibili i seguenti operatori di assegnamento:
`+=   -=   *=   /=   %=   <<=   >>=   &=   |=   ^=`
Anche le espressioni binarie composte con questi operatori restituiscono un valore, e possono essere usate in altre espressioni.

Operatori di incremento e decremento. Gli operatori unari `++` e `--` si applicano solo a variabili, aggiungono o sottraggono 1 dal loro argomento e possono essere usati come operatori prefissi o postfissi. Se `a` e `b` valgono entrambe 5, le istruzioni

```
c = ++a;
d = b++;
```

hanno l'effetto di incrementare `a` e assegnare il nuovo valore, 6, a `c`; assegnare a `d` il valore di `b` e poi incrementare `b`.

Operatore ternario. Il C ha ancora un altro operatore "strano", l'operatore ternario "?:" che consente di scrivere *espressioni condizionali.*
Nell'espressione

espr1 ? *espr2* : *espr3*

viene valutata per prima *espr1*: se il valore non è 0, il valore è quello di *espr2*, altrimenti è quello di *espr3*. Il tipo è il più generale dei tipi delle due espressioni dopo il `?`.

Istruzioni (statements)
In C qualunque espressione seguita da `;` è una istruzione legale; ma il flusso del programma procede in modo sequenziale, e senza istruzioni che lo modifichino il linguaggio non potrebbe esprimere altro che il calcolo di espressioni.
Una o più istruzioni fra parentesi graffe { e } formano una *istruzione composta* o *blocco*, che può comparire in ogni contesto in cui è lecito scrivere una istruzione semplice. Il blocco non è seguito da "`;`".

Istruzione **if - else**

```
if ( espressione )
        istruz_1
else
        istruz_2
```

La parte `else` è opzionale. Se il valore di *espressione* è diverso da 0 viene eseguita *istruz_1*, altrimenti, se presente, *istruz_2*.

Le due istruzioni possono essere istruzioni semplici, blocchi o magari essere a loro volta istruzioni `if - else`.

La parte `else` si riferisce SEMPRE all' `if` precedente più vicino, indipendentemente dalle intenzioni del programmatore. In caso di pericolo, usare le parentesi graffe { e } !

Nel caso di più `if - else` in cascata per una scelta multipla è usuale scrivere l'istruzione che ne risulta come:

```
if ( espressione_1 )
        istruz_1
else if ( espressione_2 )
        istruz_2
else if ( espressione_3 )
        istruz_3
```

Istruzione **switch**

```
switch ( espressione )
{
     case espr-cost_1 : istruzioni_1
     case espr-cost_2 : istruzioni_2
     ...
     default: istruzioni
}
```

È una istruzione di scelta multipla: calcola il valore della *espressione* (intera) e se questo corrisponde ad uno elencato nei `case`, trasferisce il controllo alla istruzione che segue; se tale valore non compare ed è presente la *label* (etichetta) `default:`, il controllo passa alla istruzione seguente. Il flusso del programma passa poi attraverso tutte le rimanenti istruzioni, a meno che non venga usata l'istruzione `break`, che passa il controllo alla prima istruzione dopo il blocco `switch`.

Cicli: `while`, `do - while`, `for`

`while ( `*espressione*` )`
    istruzione

L' *espressione* viene valutata: se è diversa da 0, viene eseguita l' *istruzione*. La cosa si ripete fino a quando il valore è 0. Analogo a `while` è il costrutto `do - while`:

`do`
    istruzione
`while ( `*espressione*` );`

Viene eseguita l' *istruzione*, poi viene valutata l' *espressione*, e se il valore è diverso da 0 la cosa si ripete.
Il costrutto più versatile per effettuare iterazioni è quello `for`.

`for ( `*espr_1*` ; `*espr_2*` ; `*espr_3*` )`
    istruzione

espr_1 e *espr_3* sono normalmente degli assegnamenti, separati da virgole, che rispettivamente assegnano i valori iniziali e aggiornano i valori di qualche variabile, mentre *espr_2* è una espressione relazionale.
Per prima viene valutata *espr_1*, poi la condizione *espr_2*; se questa è vera, si valuta *istruzione*, poi le reinizializzazioni *espr_3* e di nuovo la condizione. Il ciclo si ripete finché *espr_2* rimane vera.

Se `i` e `j` sono variabili intere, quali numeri vengono stampati dalla seguente istruzione?

```
for (i=10, j=1; i+j < 100; i+=10, j*=2)
    printf("%d ", i+j);
```

Istruzioni `break` e `continue`

L'istruzione `break` consente, oltre che di uscire dal corpo di una istruzione `switch`, anche di abbandonare l'esecuzione del ciclo in cui compare.
L'istruzione `continue`, di uso più raro, termina l'esecuzione della iterazione corrente del ciclo e provoca l'esecuzione immediata della parte di controllo.

`goto` e labels

In C una istruzione può essere preceduta da una etichetta, che ha la forma di un identificatore seguito da `:` .
L'istruzione `goto` *label*; trasferisce il controllo alla istruzione che segue l'etichetta *label*.
È buona educazione non usare il `goto` a meno di condizioni assolutamente eccezionali.

Dati aggregati – strutture

```
#include <stdio.h>
struct pt2
{
   double x, y;
}  p1 = { 3. , 5. };
struct pt2 somma(struct pt2 a, struct pt2 b);
struct pt2 scala(double, struct pt2);
struct pt2 makept2(double , double);
int        printpt2(struct pt2);
int main( )
{
   struct pt2 p2 = { 7., -1. }, p3;
   printf("p1:          "); printpt2(p1);
   printf("p2:          "); printpt2(p2);
   printf("p2 - p1:     ");
   printpt2(somma(p2,scala(-1.,p1)));
   printf("punto medio: ");
   printpt2(scala(.5,somma(p2,p1)));
   p3 = makept2( 2, 5 );
   printf("p3:          "); printpt2(p3);
   printf("baricentro:  ");
   printpt2(scala(1./3,
             somma(p3,somma(p2,p1))));
   return 0;
}

struct pt2 somma(struct pt2 a, struct pt2 b)
{
   struct pt2 loc;
   loc.x = a.x + b.x;  loc.y = a.y + b.y;
   return loc;
}
struct pt2 scala(double k, struct pt2 a)
{
   struct pt2 loc;
   loc.x = k * a.x;  loc.y = k * a.y;
   return loc;
}
struct pt2 makept2( double x, double y )
{
   struct pt2 loc;
   loc.x = x;  loc.y = y;
   return loc;
}
int printpt2( struct pt2 a )
{
   return printf("(% 4.2lf,% 4.2lf)\n",
                 a.x,a.y);
}
```

Questo programma stampa:

```
p1:          ( 3.00, 5.00)
p2:          ( 7.00,-1.00)
p2 - p1:     ( 4.00,-6.00)
punto medio: ( 5.00, 2.00)
p3:          ( 2.00, 5.00)
baricentro:  ( 4.00, 3.00)
```

Le variabili `p1`,`p2` e `p3` sono strutture, composte da due campi, accessibili come *var*`.x` e *var*`.y`.

È possibile scrivere funzioni che accettano parametri `struct pt2` e restituiscono valori dello stesso tipo, semplificando di molto la scrittura del programma.
A differenza dei tipi elementari, il C non consente di scrivere costanti strutturate, tranne che nella inizializzazione di una variabile; `p1` e `p2` sono inizializzate nella loro dichiarazione; a `p3` viene assegnato il valore restituito dalla chiamata di funzione `makept2( 2, 5 )`.
Nella definizione di una struttura i campi non sono necessariamente omogenei e possono essere di qualunque tipo già definito.
Il nome della struttura è noto al compilatore solo dopo che la dichiarazione è stata valutata, e quindi non è possile dichiarare una struttura ricorsiva (infinita) come la seguente:

```
struct NoGood
{
   int           i;
   struct NoGood x;
};
```

È però possibile, come vedremo, dichiarare una struttura in cui un campo sia un *puntatore* a una variabile dello stesso tipo:

```
struct Good
{
   int           i;
   struct Good  *x;
};
```

Dati aggregati – vettori
Il C consente di definire un vettore di dati di un qualunque tipo con la sintassi
tipo identif [*numelem*] ;
Al momento della definizione il vettore può anche essere inizializzato, come nell' esempio:

```
int myvec[5] = { 1,2,3,4,5 };
```

Se sono forniti meno valori del numero di elementi del vettore gli altri sono posti a `0`, se ve ne sono di più viene segnalato un errore.
È possibile anche una inizializzazione del tipo

```
char spazi[] = { ' ', '\n', '\t' };
```

In questo caso viene allocata esattamente la dimensione necessaria per contenere gli inizializzatori. Una espressione possibile (costante) per ottenere il numero di elementi del vettore `spazi` è `sizeof(spazi)/sizeof(char)`.
Il linguaggio non prevede una sintassi particolare per dichiarare vettori con più di un indice; è però possibile rappresentare una matrice 3×5 di `float` come

```
float mymat[3][5];
```

Ci si riferirà agli elementi con `mymat[i][j]` con $0 \leq \texttt{i} < 3$ e $0 \leq \texttt{j} < 5$.

Dati aggregati – union

```
#include  <stdio.h>
#include  <stdlib.h>
#include   <time.h>
// #define GNU
enum _NUMTYP_ { _INT_, _DBL_ };
union _NUM_  {
    int i;
    double x;
};
struct _MyNUM_ {
    enum  _NUMTYP_ tipo;
    union _NUM_    v;
};
void prtMyNUM( struct _MyNUM_ w ) {
         if (w.tipo==_INT_ )  printf("%s\t%d\n", "int   ", w.v.i);
    else if (w.tipo==_DBL_ )  printf("%s\t%f\n", "double", w.v.x);
    else                      printf("errore\n");
}
struct _MyNUM_ pippo[3]
#if defined GNU
=  {   { _DBL_,  {.x = 3.14}  },   { _INT_,  {.i = 1024 }    },  { _DBL_,  {.x = 6.28}  }
#endif
;

int main() {
#if !defined GNU
    int i;
    srand(time(0)); i = rand() % 3;
    pippo[i].tipo = _DBL_;  pippo[i].v.x = 3.14;  i = (i+1)%3;
    pippo[i].tipo = _INT_;  pippo[i].v.i = 2048;   i = (i+1)%3;
    pippo[i].tipo = _DBL_;  pippo[i].v.x = 6.28;
#endif
    prtMyNUM( pippo[0] ); prtMyNUM( pippo[1] ); prtMyNUM( pippo[2] );
    return 0;
}
```

La dichiarazione del tipo `union _NUM_` ha la stessa sintassi della dichiarazione di una `struct`; una variabile `a` di questo tipo ha due campi. `a.x` e `a.i`, che però *non esistono contemporaneamente*: dato che le aree di memoria dei vari campi **sono sovrapposte**, solo uno di essi può contenere un valore valido.
Per questo motivo, l'uso più frequente del tipo `union` è all'interno di tipi `struct` come in

```
struct _MyNUM_ {
    enum  _NUMTYP_ tipo;
    union _NUM_    val;
};
```

Per stampare una variabile w di tipo struct _MyNUM_ se ne consulta il campo w.tipo e si stampa il campo w.i oppure il campo w.x.
L'inizializzazione statica degli elementi di pippo è attivata solo se è definito il simbolo GNU; è accettata dal compilatore GNU ma non, ad esempio, dal Visual C Microsoft.

Scope (dominio) delle variabili

```
#include <stdio.h>
int x=5, y=10;
void prova( );   /* prototipo */
int main( )
{
  printf("main(1): %5d %5d\n", x, y);
  prova();
  printf("main(2): %5d %5d\n", x, y);
  {
    double x=3.14, y=6.28;
    printf("main(3): %5.2lf %5.2lf\n",x,y);
  }
  printf("main(4): %5d %5d\n", x, y);
}
void prova( )
{
   int y;
   y = x*x;   x=15;
   printf("prova:   %5d %5d\n", x, y);
}
```

Questo programma stampa:

```
main(1):     5    10
prova:      15    25
main(2):    15    10
main(3):  3.14  6.28
main(4):    15    10
```

La funzione `prova()` definisce una sua variabile `y`, e vede la variabile `x` definita globalmente.
Lo scope di `y` come variabile globale è dato dalla funzione `main()`, tranne che entro il blocco.
Lo scope di `x` comprende la funzione `main()`, fuori dal blocco, e la funzione `prova()`.
Se definizioni di funzioni e dichiarazioni di variabili *globali* si alternano, lo scope di una variabile globale è dato da tutte le funzioni la cui definizione segue nel file la dichiarazione della variabile.

Chiamata per valore

```
#include <stdio.h>
void cambia( int a )
{
printf("cambia: a = %d\n", a);
a=345;
printf("cambia: a = %d\n", a);
}
int main( )
{
     int a = 1;
     printf("main:   a = %d\n", a);
     cambia(a);
     printf("main:   a = %d\n", a);
     return 0;
}
```

L' output è:

```
main:  a = 1
cambia:a = 1
cambia:a = 345
main:  a = 1
```

Nel corpo della funzione `cambia()` la variabile `a` viene modificata, ma nel `main()`, al ritorno dalla funzione, il valore di `a` non risulta alterato.
La `a`, parametro formale di `cambia()`, appartiene a tale funzione come ogni variabile definita internamente; nell'area di memoria corrispondente viene *copiato* il valore dell'argomento con cui `cambia()` è stata chiamata.

Variabili statiche e automatiche, ricorsione
Le variabili definite dentro una funzione possono essere dichiarate `static` oppure `auto`; se nessuno dei due qualificatori è presente sono implicitamente automatiche. Vediamo su un esempio:

```
#include <stdio.h>
void pippo( )
{
     int i = 0;
     static int j = 0;
     printf( "i=%d, j=%d\n", ++i, ++j );
}
int main( )
{
     pippo(); pippo(); pippo(); return 0;
}
```

che stampa:

```
i=1, j=1
i=1, j=2
i=1, j=3
```

La variabile `i` viene inizializzata ogni volta che viene chiamata la funzione `pippo()`, mentre `j` viene posta a zero una volta sola. Ha come scope solo il corpo della funzione dove è dichiarata, ma per il resto si comporta come una variabile definita esternamente.

La necessità delle variabili automatiche si apprezza meglio analizzando una funzione ricorsiva, come nel caso del fattoriale:

```
#include <stdio.h>
long fact( long x )
{
     static long i = 0;
     printf( "(i=%ld) ", ++i );
     if ( x == 0 )
        return 1;
     else
        return x * fact(x - 1);
}
int main( )
{
     long n = 4;
     printf("\nFact(%ld)=%ld\n",n,fact(n));
     n = 5;
     printf("\nFact(%ld)=%ld\n",n,fact(n));
     return 0;
}
```

Questo programma stampa:

```
(i=1) (i=2) (i=3) (i=4) (i=5)
Fact(4)=24
(i=6) (i=7) (i=8) (i=9) (i=10) (i=11)
Fact(5)=120
```

L'identificatore `x` non può corrispondere ad una locazione di memoria fissa; devono esistere simultaneamente tante variabili `x` diverse, una per ogni chiamata della funzione `fact`, perché il calcolo di `fact(5)` richiede quello di `fact(4)` e la variabile `x` della prima istanza deve contenere ancora `5` quando la seconda istanza restituisce il suo valore, `24`.

Dai vettori ai puntatori ...

```
#include <stdio.h>
void doppio( double x )
    { x *= 2; }
void doppioV( double x[], int n )
{
    while (n-- > 0)
        x[n] *= 2;
}
void prtvecd( double x[], int n )
{
    int i;
    for (i=0; i<n; i++)
        printf("%5.2lf ", x[i]);
    printf("\n");
}
int main()
{
    double a[]={ 10., 20., 30, 40. };
    int n=sizeof(a)/sizeof(a[0]);
    prtvecd(a, n);
    doppio(a[0]);   prtvecd(a, n);
    doppioV(a, n);  prtvecd(a, n);
    return 0;
}
```

L'output è:

```
10.00 20.00 30.00 40.00
10.00 20.00 30.00 40.00
20.00 40.00 60.00 80.00
```

La chiamata di `doppio(a[0])` come si vede non ha effetti sul contenuto di `a[0]`, mentre la chiamata di `doppioV()` raddoppia tutti gli elementi del vettore. Quindi, il C non passa **sempre** i parametri per valore; ma allora che tipo di dato viene passato come primo argomento a `doppioV()`? Viene passato l'*indirizzo* dell'area di memoria associata al vettore `a`, cioè l'indirizzo del suo primo elemento.
Possiamo prendere l'indirizzo di una qualunque variabile, semplice o `struct`, di un elemento di un array, cioè di un qualunque *l-value*, di qualunque cosa possa stare a sinistra di un operatore di assegnamento, premettendogli l'operatore `&`.
In particolare, possiamo modificare secondo e terzo elemento del vettore `a` con la chiamata `doppioV(&a[1],2);`.
`a`, `&a[1]` sono due valori costanti di una specie che vediamo per la prima volta, sono indirizzi.

Duale dell'operatore `&` è l'operatore unario, prefisso, di indirezione `*`. Le istruzioni

```
b = *(&(a[3]));
*(&(a[0])) = 1.;
```

sono equivalenti a

```
b = a[3];
a[0] = 1.;
```

Gli indirizzi possono essere assegnati come valori ad una variabile. Il modo per dichiarare che la variabile `intpt` può contenere l'indirizzo di un intero è consistente con il fatto che l'espressione `*intpt` sia un `int`:

```
int    *intpt;
double a;
double *vec1;
double *vec2 = &a, vec3;
```

L'oggetto puntato dalla variabile `vec1` è un numero `double` La variabile `vec2` è un puntatore a `double`, e all'inizio punta alla variabile `a`. La variabile `vec3` invece è un `double`, non un puntatore a `double`!

Confrontiamo queste due dichiarazioni e relative inizializzazioni:

```
char   stringa1[] = "black";
char   *stringa2  = "white";
```

`stringa1` è un vettore di caratteri, il compilatore alloca lo spazio per 6 caratteri e vi scrive, nell'ordine, `'b'`,...,`'k'`, `'\0'`. Quindi `stringa1` è una **costante**.
`stringa2` è un puntatore a carattere. Per la sua inizializzazione il compilatore alloca lo spazio per i 6 caratteri `'w'`,...,`'e'`, `'\0'` e scrive in `stringa2` l'indirizzo del carattere `'w'` . Il valore di `stringa2` può in seguito essere alterato, in modo da farla puntare a qualche altro carattere o array di caratteri.

Un puntatore deve essere inizializzato in modo da puntare ad un indirizzo **valido**, ottenuto via `&` da una variabile, oppure restituito dalle funzioni di sistema per la allocazione dinamica della memoria (come vedremo), oppure dovrebbe essere inizializzato a `NULL`, eventualmente controllando prima di usarlo che il suo valore sia diverso da `NULL`.

Un altro modo, più riposto, per ottenere un puntatore non buono (e trovarsi nei guai) è quello di scrivere una funzione che restituisca come valore l'indirizzo di una variabile automatica. All'uscita dalla funzione, non è garantito che il valore contenuto a quell' indirizzo sia mantenuto o, ancora peggio, può accadere che l'indirizzo sia di nuovo in uso per qualche altra variabile automatica.

```
double *somma( double x, double y )
{
    double z = x+y;
    return &z;       /* NO!!! */
}
```

Puntatori e vettori

Dato un vettore dichiarato come `int ivec[10];`, e la variabile puntatore `int *pt=ivec;` nel quadro

```
ivec+2      pt+2    &ivec[2]    &pt[2]
ivec[3]     pt[3]   *(ivec+3)   *(pt+3)
```

le espressioni della prima riga danno l'indirizzo del terzo elemento del vettore `ivec`, quelle della seconda, l'intero contenuto nel quarto elemento del vettore. La scelta del modo di scrivere è lasciata al gusto personale. Se `pt1` e `pt2` sono due puntatori ad interi che puntano ad elementi di uno stesso vettore, la loro differenza è la differenza, col segno, dei corrispondenti indici nel vettore.
Anche ai puntatori è possibile applicare gli operatori di assegnamento `+=` e `-=` e gli operatori unari di pre- e post-increment, di pre- e post-decrement, `++` e `--`.

Vediamo un esempio "molto C" dell'uso di `++` con puntatori:

```
#include <stdio.h>
void copia( char *dst, char *src )
{
    while ( *dst++ = *src++ );
}
int main()
{
    char buf1[]="Hallo!", buf2[20];
    copia(buf2, buf1);
    printf("%s\n", buf2);
    return 0;
}
```

L'istruzione `while` nella funzione `copia()` copia un carattere dall'indirizzo in `src` a quello in `dst`, incrementa di 1 i due puntatori, e se il carattere copiato è diverso da `'\0'`, esegue l'istruzione successiva, che è vuota (`;`), e ripete il ciclo.

Torniamo all'esempio della struttura `struct Good` che avevamo lasciato in sospeso.

```
struct Good
{
   int           i;
   struct Good  *x;
}  *ptr;
```

Supponiamo che `ptr` punti ad una `struct Good`; il campo `i` della variabile puntata è espresso come `(*ptr).i`, e le parentesi sono necessarie perché la precedente di `*` è inferiore; il campo `i` della variabile puntata dalla variabile puntata è espresso come `*((*ptr).x).i`, e la notazione è alquanto faticosa.
Ne esiste una abbreviazione: le due espressioni possono scriversi rispettivamente come `ptr->i` e `ptr->x->i`, con la associatività a sinistra.

Funzioni standard di input/output (stdio.h)
Le operazioni di input/output avvengono attraverso *streams* di dati. Quando un programma parte, sono già aperti gli streams `stdin` (standard input, normalmente la tastiera), `stdout` e `stderr` (standard output e standard error, normalmente il terminale). Un file su disco, o una altra periferica, può venir associato a uno stream usando la funzione `fopen` che restituisce un valore di tipo `FILE *`; questo legame viene interrotto quando lo stream viene chiuso invocando la funzione `fclose`.

```
FILE *fopen( const char *filename, const char *mode);
```

Restituisce un puntatore al file aperto, o `NULL` se l'operazione fallisce.
I valori di `mode` comprendono: `r` (read); `w` (write - azzera il contenuto); `a` (append - in coda al file); `r+` (read/write); `w+` (read/write - azzera il contenuto); `a+` (read/write - in coda al file).
Una `b` in `mode` (es. `"rb"`, `"wb"`) indica che il file è aperto in modo binario (nessuna conversione, ad esempio fra `'\n'` e i caratteri *CR LF*).
Fra una scrittura ed una lettura dello stesso file è necessario chiamare la funzione `fflush`, o una funzione di riposizionamento.

```
FILE *freopen(const char *filename, const char *mode, FILE *stream);
```

apre il file e lo associa a `stream`; restituisce `stream` in caso di successo, `NULL` altrimenti.

```
int fflush( FILE *stream );
```

Forza l'aggiornamento su disco di un file in scrittura. Restituisce `0` in caso di successo.
`fflush(NULL)` aggiorna tutti i file aperti.

```
int fclose( FILE *stream );
```

Chiude lo stream, restituisce `EOF` in caso di errori, oppure `0`.

```
int remove(const char *filename);
int rename(const char *filename, const char *newname);
```

rispettivamente cancellano il file `filename` o lo rinominano con `newname`.

Output formattato
Le funzioni del gruppo `printf` forniscono le conversioni per l'output formattato.

```
int fprintf(FILE *stream, const char *fmt, . .);
int sprintf(char *string, const char *fmt, . .);
int printf (const char *fmt, . .);
```

`fprintf` converte e scrive su `stream` i suoi parametri, convertiti sotto il controllo del formato `fmt`; restituisce il numero dei caratteri scritti o un numero negativo in caso di errore; `printf` fa lo stesso sullo stream `stdout`; `sprintf` formatta i suoi parametri e scrive sulla stringa `string` (che deve essere abbastanza grande).
Le specifiche di conversione in `fmt` iniziano con `%` e terminano con il carattere di conversione. Vediamo il significato dei caratteri di uso più frequente:

`c`	`int`	carattere singolo;
`s`	`char *`	una stringa;
`d,i`	`int`	notazione decimale;
`u`	`int`	notazione decimale senza segno;
`o`	`int`	notazione ottale;
`x,X`	`int`	notazione esadecimale;
`f`	`double`	[-]m.dddddd;
`e,E`	`double`	[-]m.ddddddE±ee;

Fra il % e il carattere di conversione possono essere specificati fra l'altro la lunghezza del campo e la precisione: `%10d` dice che il valore intero corrispondente deve essere stampato su un campo di 10 caratteri, mentre `%-10d` dice che il numero deve essere allineato a destra sul campo; `%10.6f` corrisponde ad un valore `float` da stampare con 6 cifre decimali su un campo lungo 10.

Input formattato
Le funzioni del gruppo `scanf` gestiscono le conversioni per l'input formattato.

```
int fscanf(FILE *stream, const char *fmt, . .);
int sscanf(char *string, const char *fmt, . .);
int  scanf(const char *fmt, . .);
```

`fscanf` legge da `stream` sotto il controllo di `fmt` e assegna i valori, convertiti, alle variabili puntate dai suoi argomenti. Termina quando ha esaurito `fmt`. Restituisce il numero di elementi letti con successo, oppure `EOF`.
`scanf` fa lo stesso leggendo da `stdin`;
`sscanf` legge il suo input dal parametro `string`; La stringa `fmt` contiene:
spazi bianchi, che vengono ignorati;
specifiche di conversione, introdotte da % e terminate da uno dei caratteri, `c, s, d, i, u, o, x, f, e, g`;
eventuali altri caratteri debbono essere presenti nell'input, e non vengono assegnati a nessuna variabile.
I caratteri `d, i, u, o, x` che convertono un intero in base 10, con o senza segno, in base 8 o in base 16, possono essere preceduti da `h` se l'argomento è un puntatore a `short` e da `l` se l'argomento è un puntatore a `long`; i caratteri `f, e, g` che convertono un numero reale con segno possono essere preceduti da `l` se l'argomento è un puntatore a `double` invece che a `float`, e da `L` se l'argomento è un puntatore a `long double`.

Input/output di caratteri

`int fgetc(FILE *stream);`
restituisce un carattere, oppure `EOF` .
`int fputc(int c, FILE *stream);`
scrive su `stream` il carattere `c`; restituisce il carattere o `EOF` in caso di errore.
`int ungetc(int c, FILE *stream);`
riposiziona su `stream` il carattere `c`, che sarà disponibile per la prossima lettura.

`getchar()`, senza parametri, e `putchar(c)` equivalgono a `fgetc( stdin )`, e `putchar(c, stdout)`.

`char *fgets(char *s, int n, FILE *stream);`
legge e copia in `s` al più `n-1` caratteri, o fino al carattere `'\n'`, che viene copiato. La stringa viene terminata da `'\0'`. La funzione restituisce `s` o `NULL` in caso di errore.
`int fputs(const char *s, FILE *stream);`
scrive su `stream` i caratteri della stringa `s`. Restituisce `EOF` in caso di errore.
`char *gets(char *s);` equivale a `fgets`, legge da `stdin`;
`int puts(char *s);` equivale a `fputs`, scrive su `stdout`.

Le funzioni viste fin qua sono prevalentemente usate per la manipolazione di file di testo. Vi sono poi funzioni per input e output binario (senza conversioni), e funzioni di posizionamento:

```
size_t fread(void *ptr, size_t size,
             size_t nobj, FILE *stream);
```

legge da `stream` al più `nobj` oggetti di lunghezza `size` copiandoli nell'array `ptr`; restituisce il numero degli oggetti letti, che può essere inferiore a `nobj` in caso di fine-file o altri errori.

```
size_t fwrite(void *ptr, size_t size,
              size_t nobj, FILE *stream);
```

scrive su `stream` dal vettore `ptr` un numero `nobj` di oggetti di lunghezza `size`; restituisce il numero degli oggetti scritti.

```
int fseek(FILE *stream,long offset,
                       int origin );
```

consente di posizionarsi nel file a una distanza `offset` dalla posizione di inizio file (se `origin` è `SEEK_SET`), dalla fine del file (`origin` = `SEEK_END`) o dalla posizione attuale (`origin` = `SEEK_CUR`).

`long ftell( FILE *stream );` restituisce la posizione attuale nel file, o `-1` in caso di errore.

`void rewind( FILE *stream );` riposiziona il file all'inizio.

Funzioni di errore

Per conoscere lo stato di un file sono utili le funzioni

`int feof( FILE *stream );`: restituisce un valore `!=0` se lo stream è in condizione di `EOF`;

`int ferror( FILE *stream );`: restituisce un valore `!=0` se è avvenuto un errore nelle operazioni sullo stream;

Ulteriori informazioni possono essere lette dalla variabile globale `errno` (dichiarata in `<errno.h>`). Le condizioni di errore e fine file per `stream` possono essere cancellate chiamando `clearerr(stream)`.

Controllo dei caratteri (`<ctype.h>`)

Le principali funzioni di questo gruppo e le condizioni sotto cui restituiscono un valore *vero* sono:

`isalpha(c)`	`c` è alfabetico
`isdigit(c)`	`c` è 0 . . . 9
`isalnum(c)`	= `isalpha(c)||isdigit(c)`
`islower(c)`	`c` è a . . . z
`isupper(c)`	`c` è A . . . Z
`isspace(c)`	`c` è ' ', new-line, tab, vtab, form-feed

Vi sono poi due funzioni di conversione:

`int tolower(c)` converte `c` in minuscolo;
`int toupper(c)` converte `c` in maiuscolo;

Funzioni sulle stringhe (`<string.h>`)

Le funzioni di questo gruppo che iniziano con `str` sono destinate al trattamento di stringhe C (vettori di caratteri terminati da `'\0'`). Elenchiamo le principali; `s` e `t` sono `char *`, `cs` e `ct` sono `const char *`, `n` è `size_t`.

`char *strcpy(s, ct)`
  copia `ct` in `s`; restituisce `s`;
`char *strcat(s, ct)`
  copia `ct` in coda a `s`; restituisce `s`;
`int strcmp(cs, ct)`
  `<` 0 se `cs < ct`, `>` 0 se `cs > ct`, oppure 0:

Vi sono poi le varianti `strn` delle precedenti: ad esempio `char *strncpy(s, ct, n)` copia al più `n` caratteri.

Altre funzioni utili:

`int strlen(cs)`
restituisce la lunghezza di `cs`;
`char *strstr(cs, ct)`
restituisce un puntatore alla prima occorrenza di `ct` in `cs`, oppure `NULL`;

Funzioni matematiche (`<math.h>`)

Le funzioni matematiche del C sono tutte a valori `double`, e gli argomenti reali sono anch'essi `double`. Gli angoli sono espressi in radianti.

```
sin(x)    cos(x)     tan(x)
asin(x)   acos(x)    atan(x)  atan2(y,x)
sinh(x)   cosh(x)    tanh(x)
log(x)    exp(x)     pow(x,y)
sqrt(x)   fabs(x)    ceil(x)  floor(x)
```

`pow(x,y)` calcola `x^y`, `ceil(x)` (risp. `floor(x)`) restituiscono come `double` il più grande intero `<= x` (il più piccolo intero `>= x`). Viene segnalato un errore se l'argomento non è nel dominio della funzione, o se il valore non è rappresentabile.

Funzioni di utilità(`<stdlib.h>`)

Conversione da numero a stringa:

`double atof(const char *s)`
converte `s` in un `double`;
`int atoi(const char *s)`
converte `s` in un `int`;
`long atol(const char *s)`
converte `s` in un `long`

Generazione di numeri pseudocasuali:

`int rand(void)`
fornisce un `int` fra `0` e `RAND_MAX`;
`void srand(unsigned int s)`
usa `s` come seme per una nuova sequenza.

Funzioni aritmetiche intere:

```
int   abs(int n)     |n|;
long  labs(long n)   |n|;
```

che restituiscono il valore assoluto del loro argomento.

Allocazione della memoria:

`void *malloc( size_t size )`
: fornisce il puntatore ad un'area della dimensione `size`;

`void *calloc( size_t n, size_t size )`
: fornisce il puntatore ad un vettore di `n` oggetti della dimensione `size`;

`void *realloc( void *p, size_t size )`
: modifica l'area di memoria puntata da `p`;

`void free( void *p )`
: libera l'area di memoria puntata da `p`;

Vediamo ad esempio un programma che alloca una lista di interi letti da tastiera: l'input è terminato da 0.

```
#include <stdio.h>
#include <stdlib.h>

struct Good
{
   int           i;
   struct Good  *x;
} *lista = NULL;
struct Good *add( struct Good *p, int i )
{
   struct Good *new;
   new = (struct Good *)malloc(sizeof *lista);
   new->i = i;  new->x = p;
   return new;
}
void prtlist( struct Good *p )
{
   for ( ; p; p = p->x )
       printf("%5d\n", p->i);
}
void killist( struct Good *p )
{
   struct Good *p1;
   if (!p)
   {
      killist( p-> x );
      free( p );
   }
}
int main()
{
   int i;
   for ( ; ; )
   {
       printf("i: "); scanf("%d", &i);
       lista = add( lista, i );
       if (i == 0) break;
   }
   prtlist( lista );
   killist( lista );
   return 0;
}
```

Altre funzioni interessanti:

`void`	`abort(void)`	abortisce il programma;
`void`	`exit(int i)`	termina il programma con stato `i`;

La funzione `bsearch` implementa una ricerca binaria su un vettore ordinato; la funzione `qsort` ordina un vettore con l'algoritmo di quicksort (si consiglia di cercarle sull'help).

A scopo di debug, può essere utile la *macro* `assert`, definita in `assert.h`, come nel seguente frammento:

```
assert ( x >= 0 );
y = sqrt( x );
```

Se la `x` è minore di 0, `assert` fa si che il programma venga abortito con un messaggio:

```
Assertion failed: x >= 0, file xxx.c, line 20.
```

Per eliminare la generazione di codice da parte di `assert`, si usa la direttiva di preprocessore `#define NDEBUG`.